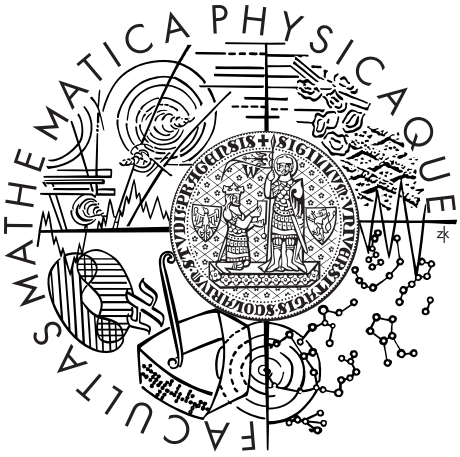
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Jakub Daniel

NetRadio

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: RNDr. Lukáš Marek
Study programme: Computer Science
Specialization: IP

Prague 2011

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In .....date ..... Signature

Název práce: NetRadio

Autor: Jakub Daniel

Katedra/Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Lukáš Marek, Katedra distribuovaných a spolehlivých systémů

Abstrakt:

V této práci je představen přehrávač internetových rádií schopný vybírat oblíbený obsah nezávisle na uživatelově zásahu, pokud to není nutné, jako řešení problému, kdy přehrávaný obsah přestane odpovídat uživatelově zájmu a uživatel je nucen ručně naladit vhodnější stanici. K dosažení tohoto cíle aplikace nepřetržitě odposlouchává všechny vložené stanice a zároveň nahrává uživatelovy preference. Pokaždé, když se změní množina právě dostupných obsahů, se může přehrávač rozhodnout, která stanice bude s největší pravděpodobností doceněna uživatelem.

Klíčová slova: internetové rádio, přehrávač, preference, automatické ladění

Title: NetRadio

Department/Institute: Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: RNDr. Lukáš Marek, Department of Distributed and Dependable Systems

Abstract:

This thesis introduces an internet radio player capable of selecting favourite content independently of the user's intervention, when it is not necessary, as a solution to a problem residing in the fact that once the station's content stops meeting user's interest he is forced to manually tune a more desirable station. To achieve this goal the application continuously monitors all stations from the list and records user's preferences. Every time there is a change in the currently available set of contents the player may decide what is the station most likely to be appreciated by the listener.

Keywords: internet radio, player, preferences, automated tuning

Acknowledgement

I would like to thank my supervisor, RNDr. Lukáš Marek, for his valuable advices and constructive criticism.

Contents

Introduction

The main goal is to introduce an internet radio player capable of selecting the most fitting station to be played from a set of stations available to the player. The selection itself will happen with only as little user's cooperation as possible. The reason here is to avoid repetitive interruptions to what the user is doing at the given time. The only phase in which an extensive user's activity is needed, and this need can be tolerated, is the initialization phase of the player in which a list of stations is created. Setting up preferences in the beginning may also request a considerable effort, but that is only temporary. Basically the application will recieve a feedback from user when it encounters new songs and other types of content. This feedback can be provided at any time and it is not obligatory to provide it for all the content. The meaning of this is to allow listener to express his preferences for individual songs, artists, stations or genres. Based on these hints the player can distinguish the favourability of the content available from every single station at any given time. In case that the identified best choice is not the currently playing station a notification may be displayed to make the listener aware of it or the station may be tuned automatically if the user wishes so.

One of the most common use cases for internet radio player is providing a music background for other user's activities. Hence it is not desired to force the listener to interrupt what he is doing to control the player. On the other hand it is not desired to force him to continue listening to content that is no longer of his interest either.

When the player cannot rely on user's intervention in case of content change which is not in favor of the user, the player has to introduce some kind of autonomous system for picking up the correct source of the audio content. Such a system has to reflect the user's needs and requirements.

The solution introduced in this project is based on recording the actions (mirroring the preferences of a listener) that lead to changing internet radio stations.

It is important to not only possess the statistics, but to be able to apply selection on the available sources as well. That requires constant scanning of the source space keeping the most favorable candidate playing.

One of the key elements of this approach incorporates means of acquisition of the sources. It is possible to extend the set of internet radio stations known to the player at very low cost without relying on user's interference. While working in today's desktop environment one often comes across several sources he might be interested in, it is possible to collect those references to new sources of media. As an example a clipboard monitoring may be used.

Once the collection is large enough to satisfy needs of the user another issue arises. It is the question how to determine the best match to the user's preferences. This project's concept employs an overall monitoring of the whole set of sources.

Chapter 1 justifies the selection of libraries used as a base of the project. It also gives reasons why certain approaches were applied. A user and programmer documentations follow in this order. Chapter 2 covers mainly the graphical user interface (`GUI`), notification system. Chapter 3 then focuses on the implementation of the project with respect to points raised in chapter Analysis, it serves as a software documentation.

The author of this thesis was not aware of any similar attempts already implemented in an existing player.

1. Analysis

1.1 Technologies

The most fundamental part of the project is undoubtedly the playback. The `Gstreamer` library was selected to provide this functionality. The decision is justified by the fact that it is capable of fetching metadata from multimedia streams and that `Gstreamer` and `GTK` are both based on `Glib` and share a similar approach in handling events. Thus their integration and communication is easier and does not need another adaptation layer. Because `Gstreamer` and `GTK`'s functionality is handled in a common main loop there is also no need to worry about synchronization between these two.

As mentioned above the `GTK` library was chosen for the `GUI`. Some of the reasons have partly been explained earlier. The collaboration with `libnotify` also benefits from use of `GTK` as it is possible to associate individual notifications with one source and user than can supress, filter or handle in any other way notifications in groups.

The use of `libnotify` is forced by absence of a better utility that would provide similar functionality. Its virtue comes from the support for responses and its low annoyance of the user.

1.2 Monitoring, playback, recording

The chosen playback library provides elements that form a so called pipeline that can fetch, process (encode, decode, trim, multiplex ...) and present data (through audio output). If any element of the pipeline fails to process its portion of data the whole pipeline is rendered inactive. To make the application robust each station needs to be provided with a separate pipeline so that no decoding failure can threaten the rest of the pipelines. An advantage of being able to operate each stream separately is gained through this concept as well.

The knowledge of what each source contains is essential and is obtained from so called metadata.

1.2.1 Metadata

One of the corner stones of the whole concept are metadata, data provided by the source along with the media stream that give additional information about the content such as name of the author or title of the song.

Metadata may or may not be provided, therefore, a fixed data structure for operating these data does not seem to serve the purpose. This is why **NetRadio** uses associative data structures of varying lenght which might not promise the highest speed performance, but with future extensions in mind it is the far most flexible solution.

When a stream of data is being read the meta information is put aside to a special collection where it is assosiated with its source. Every time there is a change in the content of that collection the change is propagated to listener through main window, notifications and statistics window. This is how the user learns the names of songs and authors. User then can provide his feedback and store his preference for the content.

From that point on, the player can tell what source is most likely to be what the user wants to listen to. A recommendetion or immediate change of station then may follow.

1.2.2 Metadata fetching

Metadata (also referred to as "tags") may be defined in various containers, as a part of the audio stream or completely separately and therefore the most general mean of tag collection is to let the appropriate plugin decode the stream and fetch the data. Any assumptions about the tags may result in complications with various formats.

A different strategy than to rely on isolated metadata accessability has to be used to counter the rise of traffic which is inevitable when all the stations are being processed on background even if they are no being played.

1.2.3 Metadata formatting

The problem here is that metadata do not comply to any standard or common sense. One such deviation from what one would expect, that could serve as an example, is that piece of metadata called *title* often holds both the name of the author and the name of the song. This project applies heuristics to solve this inconvenience.

The *title* tag can be split into pieces based on common patterns such as "**artist - song**", "**song** by **artist**" and so on. These can be matched by regular expressions for example. Another approach would be to identify substrings as names using a dictionary of all known names. This method would rely on time which it has been applied for and would be very costly. That is why the project uses only the first method.

1.2.4 Continuity of recording

A continuous recording to one file with no regards to beginnings and ends of different shows or songs is very easy to implement, but it is not terribly beneficial. On the other hand the beginnings and ends are not marked precisely, at least not by tags. That is a troubling fact and the solution to overcome it is not feasible for this thesis as it would require deep data stream analysis which would in one way or another lack its reliability, it would take a lot of resources and would be virtually impossible to implement with the rest of the project with respect to capability of performing it for multiple streams at a time. It would sound reasonable if the stations employed tags as markers, however, the tag *title* changes asynchronously with the stream, often too early.

1.3 Features that proved to be unrealizable

At the beginning there was an intention to spare the listener any unnecessary actions to present his opinion and choices. To achieve this a concept of global accelerators (hotkeys) would be a must have. It turned out that most of the desktop environments and most significantly **GNOME** do not provide this function because of one simple reason. It would mean that any application could register actions to happen on a specific keystroke event without the focus on its window. That would lead to one of two things:

1. There would be no guarrantee that a keystroke would have one exclusive meaning and would not affect multiple applications.

2. The applications would race to register keystrokes for their own use. Thus breaking their function in other applications.

Stealing focus for one application would solve both of these problems as the span of the accelerator's validity would restrain to a single application at a time. But it is an example of violation of the principles of **NetRadio** as it would interrupt user's activities and force him to respond in one or another way.

1.4 Design patterns and control flow

Because of the foreseen presence of sets, lists and other collections of value holders for various types the design patterns of *iterators*, *builders* and *factories* have been taken account for. In case of construction of the main window layout there can be one *factory* used to create different layouts and switch between them on demand. It is then possible to register a new layout with the factory and without an extensive code rewrite it can be used in the application.

The majority of the project falls into an event-driven code category handled in one main thread. Apart from that there are multiple threads being allocated. These threads are not exposed to the programmer as they are handled in the

`gstreamer` library. Nevertheless it is important to notice their presence as they have impact on the issues described in the chapter 4 (Known issues).

1.5 Comparison with other projects

The following projects were selected to compare with **NetRadio**'s preference system, user friendliness and recording capabilities.

RarmaRadio (`http://www.raimersoft.com/rarmaradio.aspx`)

RarmaRadio and QuickRadioPlayer are both developed by the same company and share most of the functionality important for this thesis. They differ mostly just in their graphical appearance.

At first sight RarmaRadio resembled a competitor to **NetRadio**, it proclaimed a capability to favour sources. It looked like it had very decent metadata display system. However the system of favourite sources proved to distinguish only two levels: the user can favour the station or not. This alone would pose a disadvantage compared to NetRadio. The metadata were fetched by RarmaRadio only for the playing station. Thus no suggestions about a better content could be made.

The recording on the other hand allowed for manual specification of the destination to record to as well as it was possible to record a continuous stream or separate songs based on the users liking.

The simplicity of `GUI`, another quality of **NetRadio**, was not met by RarmaRadio. Adding a new station takes a considerable amount of time especially at the very beginning of the application usage.

The system how the streams are stopped and played in RarmaRadio enables to add a smooth transition, which is not implemented in **NetRadio**. But it is also confusing as there are two or more stations pictured as playing.

Crawler Radio & MP3 Player (`http://www.mp3radio.com/`)

The Crawler Radio & MP3 Player has mature recording capabilities. It supports scheduling of recording and consecutive integrated playback of the recordings.

Especially the scheduling might have been integrated in **NetRadio** with the possibility of defining song name patterns rather than precise time and station to record from. The application would then be capable of collecting content favoured by the listener. This improvement has been discovered too late to integrate it in **NetRadio**.

To add to the ease of operability the Crawler Player includes an interface for selecting stations from a list of predefined ones. The lesson learned by **NetRadio** is that a parser of a common radio list such as `www.shoutcast.com` could be integrated in the player.

Rhythmbox (`http://www.rhythmbox.org`)

The Rhythmbox is a very impressive player with nice and clean interface. It has a support for playlists, podcasts, recording and it even has a rating system. Unfortunatelly rating works on the level of sources not the content and no autotuning nor recommendation features were detected either. The recording capability does seem to affect only offline sources, and thus it could be classified more as grabbing. Although it supports playback of internet radios it is apparent that it is not its primary purpose, it certainly is not convenient during a change of playing stations. The delay between order to play and actual commencement of playback is suprisingly long.

All of the players mentioned come with a visualisation element in their interface, **NetRadio** will not implement this functionality. It is supported by `gstreamer` and can be easily integrated in a `GTK` widget, it but is not necessary as it is an eye candy and a typical **NetRadio** user is not expected to watch the `GUI` for a long continuous period of time.

2. User interface

2.1 Controlling the application

After the first start of **NetRadio** the user is presented with a player with no stations specified. To fully exploit the benefits of all the features implemented in the program it is recommended to add multiple sources at this point. This can be done by typing uniform resource identifiers (`URI`'s) into the entry field in the main window or by simply copying them one by one to the clipboard (for example by **CTRL**+**C**) and waiting while the program verifies that the content given by the address is playable. The notification about an available source offers to add the source into the list of monitored stations.

Once there are enough stations to satisfy user's needs any of them can be selected from the list and it can be played or recorded. There are different ways to do so: by using the buttons or by right-clicking on the source and selecting the desired choice.

The next step would be to define some preferences for the playing content or for manually added entities. The listener is given the possibility to pre-define his favourite genres and artists or even concrete titles and assign preferences to them in the statistics window.

In case that the user specified automatic tuning in the preferences window the application may no work independently. After some time of using the application a need to adjust the list of stations and/or preferences will be needed and more than just this basic control will be necessary. Therefore a thorough `GUI` description follows.

2.2 Main window

The main window is responsible for telling the user all necessary information about what is currently playing and for letting the user control the player. There are two looks of the main window one of which provides only a minimal control and information about the state of the player and stations. The second one on the other hand displays a complete list of all stations with their specification of their content if the information is available (see Referenceschap:dependencies) and provides an interface to control all the functions of the player. The use of the minimal look is straightforward and thus the overall look will be given the attention.

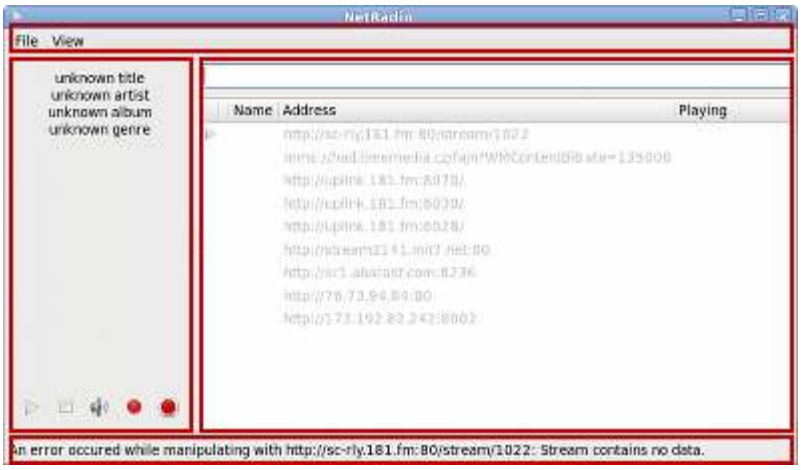


Figure 2.1: The partitioning of the overall look

The overall view is divided into four areas. The menu bar on top, the informative column on left, the selection list with a selection entry field on the right and the status bar on the bottom. To bring up other windows the menu bar can be used.

To start monitoring and playing a source an address has to be provided. That can happen semi-automatically after copying the address into the clipboard or by typing the address into the entry field.

2.3 Preferences and options

Here the user may specify the application's behaviour. Any choice he does takes effect after the application queries the incriminated value. That means that if there is a notification displayed and the listener changes the length of the period a notification is displayed for the present notification is not affected.

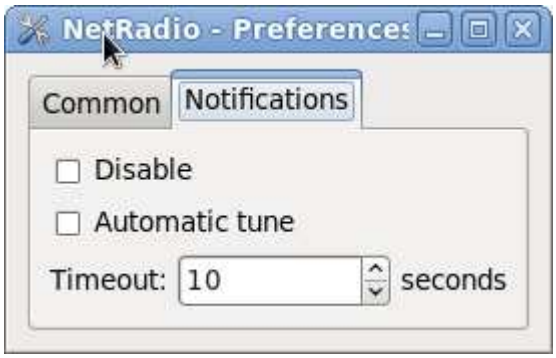


Figure 2.2: Settings

2.4 Statistics and content preferences

This window visualizes the information collected from played content. It displays songs, artists, genres, relations between all of these, amount of time spent by listening to content described by the given information. Lastly but most importantly the user's preference is displayed in this window too.

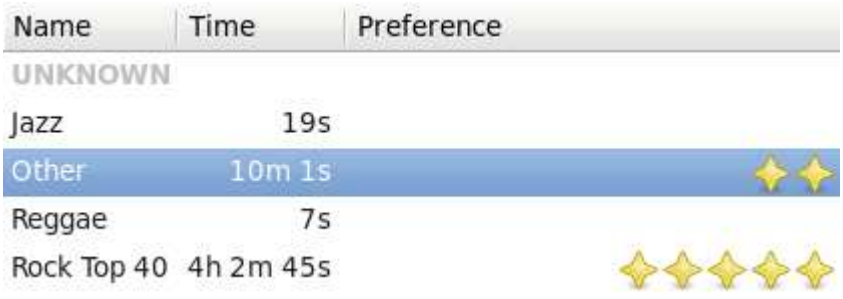


Figure 2.3: Preferences of stations

In the picture above there is a distinguishable difference between preferences of the genres. The more stars there are displayed on a row the higher the favoura-

bility is. To change the preference user may double-click on the desired row in the preference column in which case the preference increases up to five stars and then resets to zero and maybe increased again. Or the user may simply select the row and then use **CTRL**+**Y** and **CTRL**+**N** to increase and decrease the preference respectively. The preference of a currently playing content can be changed from the main window itself with the same accelerators.

Over time the statistics may grow into a size that does not allow for an easy orientation and most of the records do not have any significance for the user. A possibility to erase such records is provided in form of simple erase button that removes them all at a time.

2.5 Notifications

The last interface between the application and the user are the notifications. They are displayed regardless the state and position of the windows.

Figure 2.4: Notification

To better integrate with new desktop environments (**GNOME 3**) the notifications get attached to a status icon (Status icon). The icon then can be used to hide and show the main window.

Figure 2.5: Status icon

3. Implementation

3.1 Rough partitioning of the project

Due to the scale of the project it has to be divided into logical layers. One of which is the low level player layer. The rest of the project resides in the `GUI` layer.

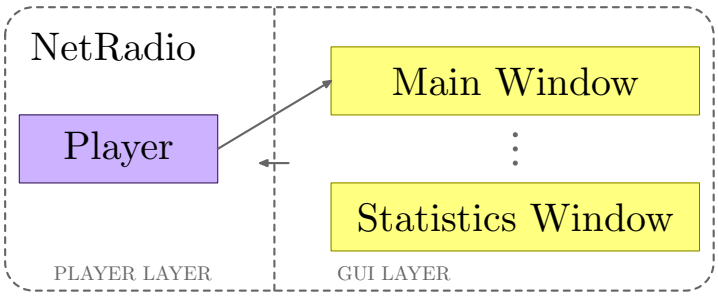


Figure 3.1: The project architecture

The figure above describes communication flow between the two layers. It is inevitable to implement `GUI`-aware interface in the player. This means that the player provides a callback interface for broadcasting data in an agreed form.

What is necessary on the other hand is that the code responsible for handling media streams and metadata is not limited by the user interface and even that the user interface is not relaying on the implementation of the player.

To simplify the communication the player often responds to the main window which itself distributes information to different parts of the `GUI`. That allows for the rest of the `GUI` to concern itself only with an appropriate narrowing of the interface as the main window can perform necessary translations. This is not mandatory and in some cases it is not desirable. It is designed exclusively for data transformations that do not belong among responsibilities of a target part of the program.

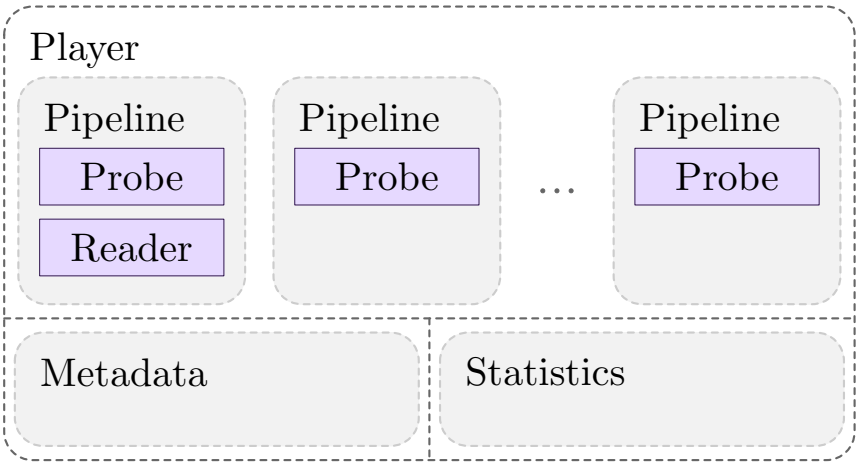


Figure 3.2: The player architecture

3.2 Detailed view of the player layer

In the picture above the terms **Pipeline**, **Probe**, **Reader** denote an object associated with one `URI` which acquires and further processes data, an object aimed at metadata fetching, an object aimed at media stream handling and playing, respectively.

The reason why only one of the pipelines in the picture has the reader object in it is to emphasize the fact that only one pipeline is playing at any given time. It could as well display a **Recorder** object. These two processing units are always part of the **Pipeline**, but they are not always activated. Unlike the **Reader** units the **Recorder** units may be active simultaneously in a higher number.

To avoid large and repetitive deallocations and reallocations of resources and structures one pipeline is created for a source at its addition to the system and it is destroyed once the source is dropped. The playback, recording and monitoring capabilities are thus united under one object.

3.3 Statistics and its role

Throughout the run time of the application all sort if illustrative information is available to the player. Those that are human readable and might meet the users interests are recorded and displayed in a standalone window. To give a

user better idea about what he might incline to, an amount of time spent on listening to a specific content is added to the records. The most significant part of the statistics held by the application, however, are undoubtedly the preferences, which are specified by user.

The preference of any entity in statistics can have values between zero and five stars. Each time a change of content occurs, the station that the change happened at is given a preference number. The station with the highest number is considered the best choice for the user and is recommended to be tuned if not playing.

The content is identified in any point in time by a set of metadata. To determine what value of preference applies to the content it is necessary to compare the set with all records in statistics with preference specified. In case that the record is a generalization of the metadata set then its value of preference is taken into account for the associated station, the highest of all the values is then the final preference of the station.

3.4 Holding information

The statistics, preferences and the station list are desirable to last between two independent runs of the application. This is achieved with a configuration file set in `XML` formated file `CONFIG/netradio/profile`.

(`CONFIG` symbolizes the default user's configuration directory)

When the application starts it parses the profile file and stores the preferences, sources and statistics in its inner data structures. Any changes to the file beyond the startup phase of the application are not reflected in the application and they are not taken account for. In case there is no configuration file present it is created. The profile is updated only once per run, after the application is exited.

The preferences are stored in `Preference` class in actual `GTK` objects as there is no reason to duplicate the values. On the other hand the statistics are stored pri-

marily in `Player` in their raw form, but they are also copied into the `Statistics` class where they are split, grouped or transformed into human readable and clear format. This minimizes the scale of one update of lists of artists, songs, stations and genres displayed to the user. The transformed information cannot be updated easily and it is required to remove it and replace it by a newly transformed information.

The shortlasting metadata are stored in property sets without a fixed size. It is an alternative to the approach known from `SQL` tables where undefined values are also needed. Although this particular implementation enables extensions.

3.5 Recording

Recording can be triggered or ceased by a user, but the rest of the process is handled automatically. The application itself decides on the destination where to record to according to what is known about the source at the time of the commence of recording. A directory system is created to represent a simple music library. Two patterns are used to provide user with the best experience:

- `MUSIC/byauthor/AUTHORNAME/ALBUMNAME/SONGNAME`
- `MUSIC/bystation/STATIONNAME/SONGNAME`

(`MUSIC` symbolizes the default user's multimedia directory for music)

The recording process is reinitialized everytime there is a metadata activity signifying a change of content. Thus each logical part of the content gets recorded into its proper destination.

All files are encoded as `ogg vorbis` files.

3.6 Scheduler

In a not so rare case of multiple stations it is likely that the program will have to download and process great amount of data. It is reasonable that the pipelines

that are actively playing or recording the content consume as many resources as they need. The expense of pipelines in monitoring state on the other hand is questionable. There is no principial excuse to download the whole stream all the time when the player is interested only in the metadata.

A problem arises from the fact that there is no generally appliable approach to fetching metadata from the stream. Not all audio containers allow for separate metadata storage which would lead to a simple way of retrieving the metadata on demand. Based on the lack of contracts ensuring this capability a need of a better control over the pipelines emerges. The goal divides into two actually. The first of them being the low traffic and the second being the correctness of metadata retrieval with a delay between two changes as little as possible. Each of these two aims tend to cancel out the other, however.

The solution implemented into this project makes use of a scheduler which ensures that every monitoring pipeline does get to download the stream ever so often so that the collected metadata are not outdated. The pipelines then have to stop monitoring right after they have completed their collection of important tags and wait for the scheduler to revive them again.

The very implementation of the scheduler is then straightforward using a synchronized queue overseen by a separate thread which operates on such frequency so that the user cannot tell the difference from having the pipelines active all the time other way than by noticing a drop in the network traffic.

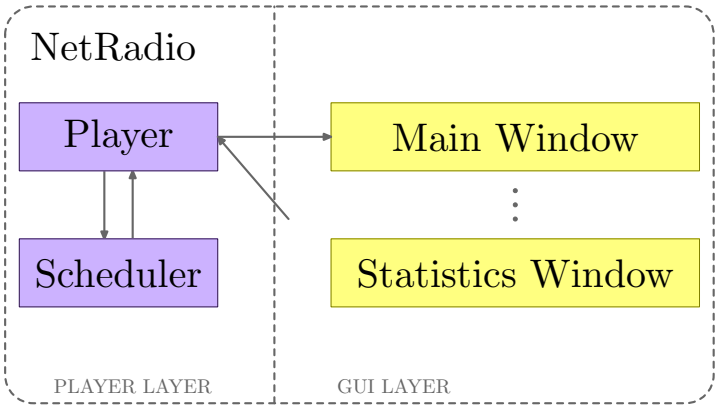


Figure 3.3: Partitioning extended with scheduler

3.7 Availability, dependencies and compilation

The project is available from `https://github.com/jd823592/NetRadio` in form of source files. Necessary libraries are not provided there, they count in

- `libgtkmm 2.4`
- `libnotifymm 1.0`
- `libxml++ 2.6`
- `libgstreamermm 0.10`.

The `base` and `good` plugins for `gstreamer` are also necessary. To be able to decode various common but possibly nonfree formats of streams the `bad` and `ugly` plugins would be needed as well. The code is written in `C++0x` which implies a need for a compiler that supports it. At the time of the project developement the `C++0x` features were not yet fully implemented in compilers. Thus not all of them were at my disposal (namely range-based for).

The project is accompanied with a `Makefile` and can be easily compiled with `make` without any additional arguments after all the dependencies are met.

4. Known issues

Unfortunatelly it proved to be impossible to avoid complications throughout the whole developement. After the introduction of the scheduler a thread leak appeared in the application causing it to fail playing, freeze gui or even fail. After a thorough investigation with use of `gdb`, `valgrind` and other means of debugging it can be stated with confidence that this issue originates in the library used for audio playback or in its poor documentation that does not specify concrete handling of elements when it is not intuitive.

The reason why the application runs out of threads is that the playback library (`gstreamer`) or its `C++` wrapper does not free allocated resources once done using them as it is stated in the documentation.

After a pipeline is brought to `STATE_NULL`, though it's expected to free all its resources, it still has threads handling the `Gst::Pad` elements around. The number of these threads increases with the number of transitions from `STATE_PLAYING` to `STATE_NULL` and back.

These transitions are necessary to lower the traffic that might bother the user if a great number of stations is monitored. To disable the downloading process the element has to be stopped and this is the only way to do so.

It is possible to disable the scheduler at compile time, however, by defining a `NETRADIO_DISABLE_SCHEDULER` macro.

There seems to be a difference in behaviour on different platforms when it comes to notifications. The sole deviation in this functionality has been encountered only on Ubuntu. Its `libnotify` package handles respondable notifications as dialog windows which do not add to the user's comfort as they pop up over other windows and steal the focus. These dialogs also do not disappear after time and need to be manually closed.

The application has been tested on Debian, Ubuntu, Fedora and Arch linux distri-

butions. Always the most recent release available at the time of the development was used.

4.1 Troubleshooting

If the program fails to play content from stations, displays error messages in status bar or fails on a library crash `gst-inspect` may display valuable information. It is then necessary to verify that no reasonable package is blacklisted (`gst-inspect -b`). Further more `GST_DEBUG=1...5` environment variable can be used to force informative messages to be printed by `gstreamer`.

Conclusion

Although all of the goals delimited in the thesis assignment and described in the introduction have been achieved, there is still space for improvements that were of a greater scale than what the project could possibly handle. **NetRadio** is a proof of a concept that is not very common in players and is basically well deployable as a final product for every day use. In its current state it is limited by the scheduling inconvenience, rendering the scheduling inapplicable, to a number of stations that does not pose a threat to network throughput. This complication would need to be resolved in a hypothetical expansion. More advanced scheduling techniques would certainly improve the resource allocation and consumption.

The differences between audio containers, their encodings, common sense and real tag usage have demonstrated how hard it is to find a correct general solution to a radio content identification using metadata. A simple heuristical solution was provided to deal with non standard situations in tagging conventions. Complications stemming from asynchronous nature of metadata retrieval in respect to the playback itself, as for example the automatic station tuning or recording redirection to a new file, are not resolved in this thesis.

The `C++` language was chosen for the project because it promised high-level abstraction insight into the problematics, especially with the upcoming standard. The direct resource allocation and deallocation seemed to be more appropriate than garbage collection in `Java`. At the late stages of the development it turned out, however, that the `C++` wrappers were poorly maintained and documented. It would be possible to overcome some problems with the application by using directly the underlying `C` library as it is feasible to write plugins to `gstreamer` in plain `C`, `C++` is less supported when it is desirable to preserve the compact interface. Such plugins would significantly simplify some tasks and would put more control into the programmer's hands.

To further enhance the listener's experience a support for playlists and podcasts as well as a station list, parsed from a dynamic online source such as

`www.shoutcast.com`, could be implemented.

After a longer period of usage of the application it is apparent that a plugin interface could enable various profiles to be created to apply different politics for different occasions. While working the listener needs to be distracted as little as possible, but while relaxing he might find it appropriate to express his preference ever so often to improve the experience in the first scenario. Even different genres may relate to different states of mind and may fit different situations.

Regardless future extensions the application with its unique aims developed into a solid player that serves its purpose. It can fulfil listener's needs and spare him time that he would otherwise spend by searching for a pleasing content among numerous possibilities.

Bibliography

[1] *GNOME Developer Center* [online]. 2011. GLibmm reference manual. Available from WWW: <http://developer.gnome.org/glibmm/2.24>

[2] *GNOME Developer Center* [online]. 2011. Gstreamermm reference manual. Available from WWW: <http://developer.gnome.org/gstreamermm/0.10>

[3] *GNOME Developer Center* [online]. 2011. GTKmm reference manual. Available from WWW: <http://developer.gnome.org/gtkmm/2.24>

[4] *Gstreamer* [online]. 2011. Gstreamer Application Development Manual <http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/index.html>

List of abbreviations

GTK	GIMP Toolkit
GIMP	GNU Image Manipulation Program
GNU	GNU's Not Unix
GNOME	GNU Network Object Model Environment
XML	Extensible Markup Language
GUI	Graphical User Interface
URI	Uniform Resource Identifier
SQL	Structured Query Language

Attachments

1. A CD with project source code, binaries and documentation